COMPUTING INTEGRITY
INCORPORATED
60 Belvedere Avenue
Point Richmond, CA 94801-4023
510.233.5400 Sales
510-233.5444 Support
510.233.5446 Facsimile

PROGRESS SOFTWARE BUSINESS ALLIANCE
ISV Partner

Replacing A Session SuperProcedure With A Class In OOABL

26 March 2006

Progress Software has numerous references in documentation and whitepapers to the parallels between superprocedures and classes, including a fairly lengthy, if somewhat flawed example in the Getting Started: Object-Oriented Programming manual for 10.1A that shows two parallel examples, one worked using superprocedures and one worked using classes. However, what is not shown is an example in which a class replaces a superprocedure while retaining the original program structure as a set of procedures. For a local super-procedure, this substitution is simple because the class does not need to be accessible except from the procedure that instantiates it. For a session superprocedure, a different functionality is required because the internal procedures of the session superprocedure become a part of the name space of procedures that did not invoke the superprocedure.

In other object-oriented languages, one would immediately think that this requirement required a name service such as JNDI or the use of singletons. The use of a name service seems unattractive for something as “intimate” as a session superprocedure because the single-threaded nature of the Progress client[1] seems to imply that the class would be running in a different session. However, OE10.1A does not include an implementation of singletons, so what does one do?

An example is provided in this document with accompanying sample code[2] that illustrates a technique that addresses this problem by using the `FIRST-OBJECT` method of the session handle and the `NEXT-SIBLING` method on the object identified by the `FIRST-OBJECT` method[3] to locate a previously instantiated instance of the object one desires to use in common. Before going into specifics, let us consider the example application that we desire to enhance.

The all procedures example provided illustrates one typical pattern of usage of session superprocedures, i.e., as a repository for session state information that might be required anywhere in the call stack. In this example, the file `example/Util/ContextSP.p` is a superprocedure that maintains information on login date, login fiscal year and period, function name and description, and the application code. This is instantiated by `example/Test/Main.p`, which takes the role that might be played by a menu system in instantiating the session superprocedure, providing it with initial values, and then calling other functions, which use the session superprocedure to access those values as needed. In this case, two calls are made to `example/Test/Function.p`, as if it was two different programs or functions below the menu system, and a display is made to illustrate retrieval of the values from the session superprocedure. To run this example, simple unzip the sample files into a directory on the `PROPATH` and enter `run example/Test/Main.p` into the procedure editor.

[1] See Use Case for Multi-threading in the 4GL, available at http://www.cintegrity.com/downloads.html .
[2] Available at http://www.cintegrity.com/downloads.html .
[3] This technique was originally suggested by Evan Bleicher of Progress Software during an exchange which was part of the beta program for 10.1A. Mr. Bleicher’s suggested technique proposed using these constructs within the programs themselves, i.e., in the terms of the examples provided here, the `FIRST-OBJECT` and NEXT-SIBLING usage would have been in `example/Test/NewFunction.p`. This was considered unacceptable to us because it would have required a s second conversion when singletons later became available, so we developed the technique illustrated here using these methods within the constructor of a façade class.

As noted in comments in the programs, in real world usage, the values initially set in the superprocedure would be derived from a more complex context than being simply assigned fixed values. Similarly, the values actually retrieved in lower level procedures would be limited to those that were actually needed, rather than retrieving all values as is shown here for illustration. While the example is very simple, the technique that we will illustrate for replacing `example/Util/ContextSP.p` with a class is a technique that we believe will work with any more complex requirement, excepting that we are making no effort to imitate `SEARCH-SELF` versus `SEARCH-TARGET` functionality.

To substitute this super procedure with a class we will, of course, need to change the invoking code in `example/Test/Main.p`[4] from `RUN` and `SESSION:ADD-SUPER-PROCEDURE` to `NEW`. However, we will also have to insert new code into `example/ Test/Function.p` because there is nothing in the existing OOABL implementation that would provide the name space of the object to that procedure without some specific connection. This seems unfortunate, since it does imply having to modify all programs in which a session superprocedure is accessed, but our feeling is that a modification that will be usable as is in future OO transitions is a very different issue than having to insert some code that will need to be changed again when the language evolves further. Thus, in this example we are making the choice that a change that will be either exactly what we would do were singletons available now or which can be easily refactored is an acceptable level of modification because it won't have to be re-done. Some may not be willing to make this level of modification, but then they will be blocked from replacing session superprocedures and will still be blocked when singletons become available in the language.

In a true, full OO environment, what we would expect to do in this other procedure or class is to `NEW` the same object again, but because it was declared it as a singleton, the execution environment would attach us to the previously instantiated copy instead of creating a new one. This is the form of the change that we are using, i.e., the `NEW` in the function is identical to the `NEW` in the main program, but one instantiates a new object and the other attaches to an existing object.

Since we can't yet declare a singleton class directly, what we are showing here is creating two classes `com.example.Util.Common.LoginContext.cls`[5] and `com.example.Util.Common.LoginContextImpl.cls`[6], the latter being close to the singleton class we would like to have created and the former being a façade which stands in for the "real" class and provides us with the singleton behavior while passing through method calls to the "real" class. This should allow us later, when there are true singletons in the language, to replace the façade with the "real" class, add the language structure for defining a singleton, and leave all of the other programs unmodified.

The key to this technique, then, is this code from the constructor of `com.example.Util.Common.LoginContext.cls`:

[4] In order to keep the before and after versions separate, we have used NewMenu.p and NewFunction.p for the modified versions, but in practice the original procedure names would be retained.
[5] Note that as a part of implementing classes here we are switching from a naming convention in which example is a group of related functions below a point on the PROPATH to putting the class code underneath com/example. This naming convention of putting code underneath the three letter domain type, then the domain name, is reasonably standard in other OO languages and allow packages from multiple sources to appear in the same source tree without conflicting. We have also elected to put the class one level deeper in the directory hierarchy in the expectation that we will want to organize related groups of classes within Util.
[6] We are aware that the Impl name component is not usually used in this way in the Java World, but it seems such an appropriate name convention for the current situation that we are using it anyway.

```
mob_SessionObject = session:first-object.
FIND-CONTEXT:
do while valid-object( mob_SessionObject ):
  mch_ObjectName = mob_SessionObject:getClass():TypeName.
  if mch_ObjectName eq "com.example.Util.Common.LoginContextImpl"
  then do:
    cob_LoginContextImpl = cast( mob_SessionObject, "com.example.Util.Common.LoginContextImpl" ).
    leave FIND-CONTEXT.
  end.
  mob_SessionObject = mob_SessionObject:next-sibling.
end.
if not valid-object( cob_LoginContextImpl )
then do:
  cob_LoginContextImpl = new com.example.Util.Common.LoginContextImpl().
end.
```

In this code, what happens is as follows:
1. Find the first object in the session.
2. Loop while there are still valid objects.
3. Test to see if the current object has the desired name.
4. If yes, then cast the object to the local copy and leave the loop.
5. If not, get the next object, as long as additional ones are available.
6. When the loop has been finished, see if we have a valid object as desired and, if not, create a new one.

Thus, the first time this code is executed, the object will not be found and a new one will be created, but the second and following times the existing object will be found and cast to the local reference. I.e., we will end up with one instance of the façade object for each place it is referenced, but only one copy of the actual "pseudo-singleton".

While many uses of session superprocedures are such that there is no concern over cleaning up the object when the last reference to it has gone away, for purposes of illustration in this example we have included a reference counting technique for determining when there are no longer any references to the "real" object. Thus, each `NEW` of the façade object will increment the reference count of the "real" object by one and each `DELETE` of the façade object will decrement the count of the "real" object by one and delete that object if the count drops to zero. To implement this technique, one includes a `DELETE OJBECT` in the programs that `NEW` the object, providing a very clean regimen of object management. If one wanted the object to persist until the end of the session for possible later use, one would not include this code.

To run this example, simple unzip the sample files into a directory on the `PROPATH` and enter `run example/Test/NewMain.p` into the procedure editor. For the purpose of seeing how the number of objects grows and shrinks, we have included a method called ShowObjects in both `example/Test/NewMain.p` and `example/Test/NewFunction.p`, the modified versions of the programs from the procedure-only example. This method will display all objects that exist in the session at the time the procedure is executed. You will note running the example the following sequence:
1. After the initial `NEW` of LoginContext, there are two objects, LoginContext and LoginContextImpl.
2. After the `NEW` in NewFunction.p, there are three objects, two LoginContext, but only the original LoginContextImpl.
3. After the `DELETE` in NewFunction.p, we are back to the original two.
4. Steps two and three repeat for the second call.
5. After the `DELETE` in the top level procedure, no objects remain.

While this is a simple example, we hope that it will illustrate a useful technique for those wishing to begin using classes in association with their existing procedure-based code by allowing them to substitute classes for what might be considered their closest procedural counter-part, the superprocedure. This example is specifically oriented toward the problem of how to use a class in place of an existing session superprocedure and is very much not intended as an indication of how one would provide similar functionality in an object call stack, since there are better and more direct techniques for that. However, it does seem likely that this same technique will be useful for other functions in a mixed procedure-object environment and may be useful in a pure OO environment for "discover my context" type methods.